

# Recommending Code Reviewers for Proprietary Software Projects: A Large Scale Study

Dezhen Kong<sup>\*‡</sup>, Qiuyuan Chen<sup>\*‡</sup>, Lingfeng Bao<sup>\*§</sup>, Chenxing Sun<sup>†</sup>, Xin Xia<sup>\*</sup>, Shanping Li<sup>\*</sup>  
<sup>\*</sup>Zhejiang University, China, {timkong, chenqiuyuan, lingfengbao, shan}@zju.edu.cn, xin.xia@acm.org  
<sup>†</sup>Tencent, China, marssun@tencent.com

**Abstract**—Code review is an important activity in software development, which offers benefits such as improving code quality, reducing defects and distributing knowledge. Tencent, as a giant company, hosts a great number of proprietary software projects that are only open to specific internal developers. Since these proprietary projects receive up to 100,000 of newly submitted code changes per month, it is extremely needed to automatically recommend code reviewers. To this end, we first conduct an empirical study on a large scale of proprietary projects from Tencent, to understand their characteristics and how code reviewer recommendation approaches work on them. Based on the derived findings and implications, we propose a new approach named CAMP that recommends reviewers by considering their collaboration and expertise in multiple projects, to fit the context of proprietary software development. The evaluation results show that CAMP can achieve higher scores on proprietary projects across most metrics than other state-of-the-art approaches, i.e., REVFINDER, CHREV, TIE and COMMENT NETWORK and produce acceptable performance scores for more projects. In addition, we discuss the possible directions of code reviewer recommendation.

**Index Terms**—reviewer recommendation, code review, proprietary projects

## I. INTRODUCTION

Code review is an important activity in software development, which is nowadays the best practice in both open source and industrial software projects [1], [2]. The main goal of code review is to improve the overall quality of a code change<sup>1</sup>, such as reducing software defects and quality problems of source code, through manual examination done by code reviewers [3], [4]. Since the inefficiencies of traditional peer review practices, such as *code inspections* [1], which are cumbersome and time-consuming [5], many organizations have adopted a more lightweight, tool-based code review process called Modern Code Review (MCR) [6], [7], which can not only detect defects and quality problems of source code as traditional code review practices but also distribute knowledge and increase team awareness [7].

In Tencent, a giant international company, there are a large number of proprietary projects maintained in internal community. Many teams in Tencent adopt similar MCR workflow as open-source organizations, making a proportion of proprietary projects open to internal developers. Internal developers can contribute code to the projects they are concerned with, and a

submitted code change will be merged after one or more code reviewers approve it. In this context, although developers do not build open-source software, the practices in open-source development is used. Considering that there are too many code changes to be reviewed (In Tencent, there are more than 100,000 code changes submitted to their proprietary projects per month), it is still expensive and not efficient to manually assign proper reviewers for each code change [8]. Therefore, Code Reviewer Recommendation (CRR) tools are needed to automatically find proper reviewers to promote the efficiency of software development [8], [9].

Fortunately, there have been lots of studies dedicated to CRR. Many proposed approaches [8], [10]–[15] mainly consider the expertise of reviewers as a critical factor, and there are also approaches [4], [16]–[19] considering other factors such as collaboration, workload, and knowledge distribution. Most researchers evaluated their approaches on open-source projects, typically those hosted on Gerrit systems and GitHub. A small part of studies [12], [18], [20], [21] evaluated their approaches on a small number of proprietary software projects. However, little effort is paid in systematically evaluating CRR approaches on proprietary projects in a large company, and the characteristics of these projects have not been studied.

To bridge this gap, we first conduct a large-scale empirical study to investigate the characteristics of proprietary projects in Tencent, and how existing state-of-the-art CRR approaches perform on plenty of proprietary projects in a giant commercial company. Based on the derived findings and implications, we then propose a new approach to better recommend code reviewers in the context of proprietary software development. In this work, we try to answer the following research questions: **RQ1: What are the characteristics of proprietary projects in Tencent?**

We first identify 300 proprietary projects with the most pull requests in Tencent, and retain 163 accessible projects with more than 1,000 historical code changes. After retrieving their review data, we then clean our datasets (*e.g.*, removing robot reviewers and code changes with no commits) and categorize the projects according to their dominant programming languages, since we find projects written in the same language serve different applications and functionalities. Through the quantitative analysis of our collected datasets, we obtain some valuable results which can support our further study. For instance, we find that most code changes in our collected projects are only reviewed by a single person, while those

<sup>‡</sup>Work done while this author was an intern at Tencent.

<sup>§</sup>Corresponding author.

<sup>1</sup>In this paper, we use *code change* and *pull request (PR)* interchangeably.

in open-source projects are often reviewed by several or even a group of project members; most developers tend to work with their dominant collaborators and a proportion of projects are reviewed by only several developers. We also notice that 9.6% of the reviewers have worked on multiple projects (contributing or reviewing code).

### **RQ2: How do existing approaches perform on proprietary projects?**

We evaluate four existing state-of-the-art approaches, *i.e.*, REVFINDER [11], TIE [8], CHREV [12] and COMMENT NETWORK [16] on our collected datasets. The results show that for each approach, performance scores on proprietary projects vary greatly. For example, when COMMENT NETWORK is applied, the mean reciprocal rank (MRR) can range from below 0.3 to nearly 1.0. On most proprietary projects (137 out of 150, projects written in rare languages are excluded), collaboration-based approach COMMENT NETWORK achieves higher scores than other approaches in terms of almost all metrics. COMMENT NETWORK also produces the largest reviewer participation rate for most projects. And TIE, which combines text mining and file path similarity performs better than file location-based approaches REVFINDER and CHREV. We also qualitatively analyze our findings.

### **RQ3: Can we propose a new approach to better recommend proper reviewers for proprietary projects?**

On the basis of what we acquire in RQ1 and RQ2, *i.e.*, (i) considering the Collaboration as an important factor and (ii) considering developers' Multiple-Project working experience, we propose a new approach that can fit the context of proprietary development named CAMP. Roughly speaking, CAMP recommend reviewers according to developers' collaboration along with expertise on multiple projects. For collaboration, CAMP builds a collaboration network (a directed graph) of all participants in current and relative projects, then divides the network into several communities. To supplement reviewers' expertise, CAMP leverages an identifier splitting algorithm to extract common information from text and file paths of PRs in multiple relative projects. Finally CAMP builds a term list for each candidate reviewer, containing terms in the content of PRs that he/she has participated in. For an incoming PR, CAMP first recommends reviewers who are in the same communities as the author of the PR. If there are not enough reviewers, CAMP then search the whole reviewer list for proper ones measured by the relevance scores of their expertise.

We then evaluate CAMP on our collected datasets. The results show that on average, our approach outperforms baseline approaches in terms of all metrics we use. CAMP produces acceptable recommendation results for more projects, that is, fewer projects only obtain very low top-5 accuracy. Additionally, CAMP can increase the participation rate of proprietary projects and can therefore boost knowledge distribution and team awareness, as is one of the goal of code review [7].

In summary, the contributions of our work are two-fold:

- 1) To the best of our knowledge, we are the first to investigate the characteristics of proprietary projects, and the effectiveness of CRR approaches on them by a large-

scale empirical study involving 163 projects in total. We derive several findings and implications, which provides directions for our refined approach.

- 2) We propose a new approach based on our empirical study. The new approach named CAMP outperforms other baseline methods on proprietary projects also with larger reviewer participation rate. And we further discuss the direction for CRR approaches.

The remainder of the paper is organized as follows. Section II provides the background about modern code review and code reviewer recommendation. Section III and IV present our empirical study on proprietary projects at Tencent, along with derived findings and implications. Section V exhibits the details and evaluation results of our approach. Section VI discusses advantages and limitations of our methods, and also focuses on the threats to validity. Section VII provides related studies from literature and Section VIII concludes the paper.

## II. BACKGROUND

### A. Modern Code Review Practice

We briefly describe the workflow of MCR in Tencent, which is similar to that on open source platform such as GitHub. First, a developer *forks* the main repository, and makes some code changes to the forked repository. Then she can open a *pull request* to ask for adoption of code changes. Other developers, so-called *reviewers*, will be invited to review code changes automatically or manually and they can give some comments. Some privileged reviewers can approve or reject code changes. Notice that reviewers may also be robots, which do some automated tasks like continuous integration (CI) and sanity check. The approved code changes will be merged into the main repository.

Most development teams in Tencent adopt MCR. To determine code reviewers, developers maintain owner lists and necessary reviewer lists in configuration files of projects, containing developers responsible for reviewing code changes. However, on the one hand, it costs a lot of time and effort to maintain configuration files. On the other hand, developers in maintained lists just have the privilege to manage code repositories (*e.g.*, submitting code changes), but not certainly the proper reviewers. To this end, applying CRR algorithms to proprietary projects is a possible way to improve code review practice.

### B. Code Reviewer Recommendation Approaches

There have been a lot of work related to CRR, which can be considered to be evaluated on proprietary projects. A lot of approaches are expertise-based. Thongtanunam *et al.* proposed REVFINDER [11] to automatically recommend reviewers by leveraging the similarity of changed files between reviews. Reviewers who have participated in PRs that are similar to the incoming code change in terms of file paths are probably to be recommended. Xia *et al.* [8] proposed TIE, integrating file location-based approach and text mining approach to improve the performance of REVFINDER. Ouni *et al.* introduced a search-based approach REVREC [13], to find proper reviewers

based on their expertise and collaboration in past reviews. Rahman *et al.* [14] recommend reviewers considering not only the relevant cross-project work history, but also the experience of a developer in certain specialized technologies associated with a pull request. Zanjani *et al.* proposed CHREV [12] that measures a reviewer’s expertise by her review history.

As for non-expertise-based approaches, Yu *et al.* proposed an approach named COMMENT NETWORK [16], which leverages social relations between contributors and reviewers to build a collaboration network, achieving similar performance as traditional approaches. Al-Zubaidi *et al.* proposed WL-RREC [4], which leverages a multi-objective meta-heuristic algorithm to search for proper reviewers with two objective, *i.e.*, maximizing the chance of participating in a review, and minimizing the skewness of the review workload distribution among reviewers. Rebai *et al.* formulated CRR as a multi-objective search problem to balance the conflicting objectives of expertise, availability, and history of collaborations [22]. Mirsaedi *et al.* proposed CARROT [20] to mitigate turnover in the development process, which can balance expertise, workload, and knowledge distribution.

### III. ANSWER RQ1: DATA RETRIEVAL AND ANALYSIS

In this section, we present our empirical study on the review data collected from Tencent. We aim at finding the characters of pull requests and how developers participate in code review activities. The derived findings can provide supports for further study.

#### A. Data Collection

In Tencent, there are a great number of proprietary projects that are open to internal developers, and these projects desire for the functionality of code reviewer recommendation. We first identify 300 projects with the most PRs, and remove those unrelated to real repositories and documentation websites. We then retain 163 projects containing more than 1,000 historical code changes from October 2018 to July 2021.

We group the collected projects based on the primary programming language they are written in. We determine the primary programming language of a project by the most source code files written in it. Since TypeScript extends JavaScript by adding types to the language, we regard TypeScript projects as JavaScript projects. For projects that do not contain source code, typically those only storing static datasets or written in rare languages in Tencent (*e.g.*, PHP, Dart and C#), we categorize them into *Others* and do not use them in our further discussion.

Table I presents the projects used in our study. JavaScript and Java projects make great proportions (29% and 23% respectively). By the way, most of the JavaScript projects are frontend web applications, and most of the Java projects are Android applications. Table I also exhibits mean and standard deviation of some properties of selected projects, including the number of contributors ( $\#Contrib$ , the number of reviewers ( $\#Rev$ ) and the number of PR ( $\#PR$ ).

We retain the following properties for each PR:

- 1) *Review ID*. A unique integer or string.
- 2) *Created time*. The time when the review was created.
- 3) *Author or committer*. The author or committer of the review.
- 4) *Changed files*. Changed files in related commits.
- 5) *Textual content*. Description and commit messages of a PR.
- 6) *Reviewers*. Actual reviewers of the PR.

We remove PRs which are not in *merged* or *abandoned* state, since *open* changes may not be fully reviewed and get eventual results. Additionally, Among the obtained pull requests, we remove robot participants (typically those developers whose names end with `bot`) and those reviewers in *pending* state. We also remove those PRs with no reviewer or changed files. Finally, we sort reviews for each project in chronological of their created time.

#### B. Pull Requests in Proprietary Projects

Since the high complexity of textual content of PRs, we first concern the number of reviewers and changed files in a PR. We define the following two properties applied to a project:

- 1)  $\#Rev_{pr}$ : Mean count of reviewers in a PR, which can be computed as:

$$\#Rev_{pr}(P) = \frac{1}{|P|} \sum_{pr \in P} ReviewerCount(pr) \quad (1)$$

- 2)  $\#Files_{pr}$ : Mean count of changed files in a PR, which can be computed as:

$$\#Files_{pr}(P) = \frac{1}{|P|} \sum_{pr \in P} FileCount(pr) \quad (2)$$

where  $P$  is a project, and  $|\cdot|$  is the number of PRs in a project.

We first compute  $\#Rev_{pr}$  for each project. Table II shows the distribution of  $\#Rev_{pr}$  for each programming language. Medians of  $\#Rev_{pr}$  is near 1.0, indicating that in most projects, code changes are just reviewed by one person on average, while by two or more developers in a small proportion of projects. This probably results from the review policy at Tencent: *a code change can be merged if one necessary reviewer approves it*. Here *necessary reviewers* are mainly designated by the configuration files of a project.

Compared with Tencent, the situation in other commercial companies is different. Rigby *et al.* [6] has reported that in AMD, the median number of each review is 2, and this value is 3 or 4 in some projects of Microsoft. On the other hand, in open-source projects, developers can participate in reviews they are interested in. For example, on OpenStack<sup>2</sup>, a code change is reviewed by over three reviewers on average, and up to 40.

<sup>2</sup>We retrieve code review data between August 2020 and January 2021 from <https://review.opendev.org> using Gerrit REST API.

TABLE I  
DETAILS OF PROPRIETARY PROJECTS USED IN OUR STUDY, GROUPED BY LANGUAGE.

Language	# Project	#Contrib (mean ± std)	#Rev (mean ± std)	#PR (mean ± std)
JavaScript (JS)	48	27.0 ± 21.8	34.0 ± 24.6	1993.8 ± 1411.3
Java	38	33.0 ± 36.8	41.7 ± 35.4	2998.7 ± 4620.2
C++	19	43.5 ± 42.5	52.5 ± 41.9	1819.2 ± 1149.7
Python (Py)	9	28.8 ± 29.0	39.5 ± 44.8	1306.0 ± 130.2
Go	19	40.0 ± 35.7	40.3 ± 36.8	2359.8 ± 1300.9
Objective-C (OC)	17	34.8 ± 46.7	40.8 ± 44.3	4099.1 ± 7267.7
Others	13	–	–	–

\* #Contrib, #Rev and #PR stands for the number of contributors, reviewers and PRs, respectively.

TABLE II  
DISTRIBUTION OF #Rev<sub>pr</sub> (PROJECTS ARE GROUP BY LANGUAGE).

Language	mean	std	min	median	max
JavaScript (JS)	1.22	0.34	1.00	1.02	2.29
Java	1.10	0.19	1.00	1.02	1.81
C++	1.15	0.28	1.00	1.02	2.03
Python (Py)	1.04	0.09	1.00	1.00	1.30
Go	1.35	0.43	1.00	1.06	2.07
Objective-C (OC)	1.12	0.20	1.01	1.04	1.82

TABLE III  
DISTRIBUTION OF #Files<sub>pr</sub> (PROJECTS ARE GROUPED BY LANGUAGE).

Language	mean	std	min	median	max
JavaScript (JS)	9.36	4.65	3.56	7.91	29.07
Java	12.41	5.39	1.73	11.84	24.40
C++	11.52	7.27	2.89	10.75	28.26
Python (Py)	6.54	3.38	1.60	5.14	13.15
Go	10.79	7.51	4.11	7.90	37.32
Objective-C (OC)	14.66	6.67	5.72	14.66	31.21

**Finding 1.** On most proprietary projects at Tencent, code changes are reviewed by only one developer on average.

We then count the distribution of #Files<sub>pr</sub>, and the result is presented in Table III. All groups' median #Files<sub>pr</sub> values are below 15, which means that only few files are changed in most pull requests. However, among all code changes, 2% contain more than 100 files and are difficult for reviewers to inspect. The largest number of contained files is above 2,000. Some PRs with more than 500 but less than 1,000 files are not very complex in fact (e.g., changing a class name that affects many files with reference to it). Anyway, it is burden for some CRR approaches to deal with these huge PRs.

**Finding 2.** Most code changes do not contain too many files, but a small proportion contain a large number of files.

### C. Reviewer Participation in Proprietary Projects

For characteristics of reviewer participation, we first investigate the diversity of reviewers in a project. We measure the

diversity of reviewers (*DoR*) for each project using Shannon's entropy [23], which can be computed as follows:

$$DoR(P) = - \sum_{s \in rc(P)} p(s) \cdot \ln p(s) \quad (3)$$

where  $P$  is the target project,  $rc(\cdot)$  denotes the collection of existing reviewer combinations of  $P$ , and  $p(\cdot)$  stands for probability of a reviewer combination. For example, if 5% of the code changes in  $P$  are co-reviewed by  $A$  and  $B$ , and 4% are reviewed by  $A$  only, then  $p(\{A, B\})$  and  $p(\{A\})$  are 0.05 and 0.04, respectively.

We discover that on each group, *DoR* values vary from project to project. In our datasets, several projects are reviewed just by one or two developers, therefore their *DoR* values are relatively small (often less than 1.0). We also look into the projects whose *DoR* values are below 2.0 and found that those projects usually have dominant reviewers, i.e., reviewed by several participants in turn. Except these dominant ones, others usually just review a small part of code changes.

**Finding 3.** *DoR* values vary from project to project, and there are a proportion of projects have dominant reviewers.

Since many projects are written in the same language or maintained by the same team, it is interesting to investigate whether they share developers. Among 3,967 human reviewers in all internal projects, 382 have participated in over two projects, and 98 have participated in over four projects. We also find that many of those developers participate in a group of projects. For example, developers  $D_1$ ,  $D_2$  and  $D_3$  all contribute code or review others' code changes in  $P_1$ ,  $P_2$  and  $P_3$ , we say that  $\{P_1, P_2, P_3\}$  are a group of frequently appearing projects. We leverage Apriori algorithm [24] to generate frequently appearing projects, finally resulting in 13 groups of projects (involving 98 projects in total) that share common participants.

**Finding 4.** Nearly 10% of the reviewers have worked in multiple projects.

Since we have observed that some developers tend to review certain colleagues’ code changes, or submit code changes that are often reviewed by specific colleagues, we use *dominant collaborator* to describe the phenomenon: if a developer  $D_a$  has reviewed more than half of the code changes contributed by developer  $D_b$ , then  $D_a$  is  $D_b$ ’s *dominant collaborator*, and vice versa. We compute the proportion of developers with dominant collaborators (*DDC*) for each project. The result is that over 50% of the total developers in our collected projects are *DDC*. We only count the developers who have contributed or reviewed more than 10 code changes.

**Finding 5.** Over 50% of the developers have dominant collaborators.

#### IV. ANSWER RQ2: EVALUATING CRR APPROACHES

##### A. Evaluation Metrics

In this section, we mainly concern whether existing approaches can perform well on proprietary projects, and how differently the existing approaches perform.

To evaluate existing methods on collected projects, we use four metrics widely adopted by recommendation system community, *i.e.*, top-k accuracy, MRR, precision and recall. Compared with our work, studies [8], [11] only consider top-k accuracy and MRR as performance metrics, while other studies such as [4], [12], [16] use precision and recall. These evaluation metrics are described as follows.

**Top-k accuracy** is the percentage of reviews where their ground truth code reviewers are ranked in the top  $k$  positions in the returned ranked list of reviewers. It can be calculated as follows:

$$\text{Top-k Accuracy} = \frac{1}{|S_R|} \sum_{r \in S_R} \text{isRecomm}(r, k) \quad (4)$$

where  $S_R$  is reviews in test set,  $\text{isRecomm}(r, k)$  denotes whether there exists a correct reviewer for review  $r$  in the first  $k$  positions of the recommendation list.

**Mean Reciprocal Rank (MRR)** [25] is a popular metric used in information retrieval and recommendation system. Given a query, its reciprocal rank is the multiplicative inverse of the rank of the first correct document in a rank list produced by a ranking technique. MRR can be computed as follows:

$$\text{MRR} = \frac{1}{|S_R|} \sum_{r \in S_R} \frac{1}{\text{rank}(r)} \quad (5)$$

where  $S_R$  is reviews in test set,  $\text{rank}(r)$  is the rank of the first correctly recommended code reviewer in the ranked list for review  $r$ .

**Mean Precision (MP).** Here *precision* is the fraction of recommended reviewers that are correct for a code change, which can be computed as follows:

$$\text{precision}@k = \frac{1}{|S_R|} \sum_{r \in S_R} \frac{|\text{Actual}(r) \cap \text{Recomm}(r, k)|}{k} \quad (6)$$

Mean precision averages such measures for all pull requests in the test set. In the equation above,  $S_R$  is reviewers in test set,  $\text{Actual}(r)$  is the actual reviewers of the review  $r$ , and  $\text{Recomm}(r, k)$  denotes the first  $k$  reviewers in the recommendation list of  $r$ .

**Mean Recall (MR).** Here *recall* is the fraction of ground-truth reviewers that are correctly recommended, which can be calculated as follows:

$$\text{recall}@k = \frac{1}{|S_R|} \sum_{r \in S_R} \frac{|\text{Actual}(r) \cap \text{Recomm}(r, k)|}{|\text{Actual}(r)|} \quad (7)$$

Mean recall (MR) averages such measures for all pull requests in test set. In the equation above,  $S_R$  is reviews in test set,  $\text{Actual}(r)$  is the actual reviewers of the review  $r$ , and  $\text{Recomm}(r, k)$  denotes the first  $k$  reviewers in the recommendation list of  $r$ .

##### B. Baseline Approaches

To answer RQ2, we evaluate four CRR approaches, *i.e.*, REVFINDER, CHREV, TIE, and COMMENT NETWORK on our collected projects. Four approaches are presented as follows.

**REVFINDER** [11] is a file location-based approach that recommends reviewers by leveraging the similarity of file paths. For an incoming pull request, developers who have reviewed many files similar to those in the pull request are probably to be recommended.

**CHREV** [12] is also a file location-based approach, which considers reviewers’ experience and working time on specific files. CHREV outperforms REVFINDER on many open-source projects.

**TIE** [8] combines the file location-based model and text mining model together, and is proved to be one of the most promising CRR approaches, that is, can achieve acceptable performance on all evaluated open-source projects [26].

**COMMENT NETWORK** [16] is completely a collaboration-based approach that recommends reviewers only considering their collaboration relationship. For a new PR, developers who always review the author’s code changes or are the dominant participants of the project are tend to be recommended.

Other non-expertise-based approaches are not evaluated in our experiments, including workload-aware approach WL-RREC [4], and context-aware approach CARROT [20]. The main reason is that these approaches need additional context information, which is somehow difficult to acquire in Tencent, *e.g.*, review workload of each developer (a developer may participate in many development tasks) and the background knowledge of proprietary projects, even though the information is very useful in recommending reviewers.

##### C. RQ2: Effectiveness of Existing Approaches

We evaluate four CRR approaches described above on our collected projects. Due to space limitation, we only exhibit the average of the top-k accuracy, mean precision, and mean recall for each approach in Figure 1. Since MRR and top-5 accuracy are more concerned in Tencent (five candidates are provided when the submitter of a code change tries to select

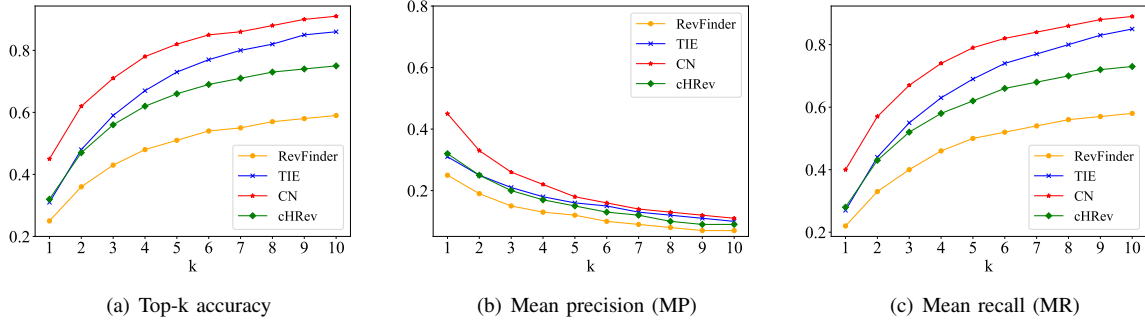


Fig. 1. Averages of top-k accuracy, MP and MR.

reviewers), we also exhibit the distribution of MRR and top-10 accuracy of each group in Figure 2 and 3, respectively.

From Figure 2 and 3, for most categories, MRR and top-5 accuracy values vary from projects to projects. On JavaScript projects, the difference is more noticeable. It is obvious that on some projects, the top-5 accuracy is still very low (*i.e.*, less than 0.3), which is somewhat not acceptable in practice.

**Finding 6.** Performance of the four evaluated approaches are various among different projects.

To investigate the reason behind various performance metrics, we use intuitive properties of collected projects, *e.g.*,  $\#PR$ ,  $\#Contrib$ , and the review frequency (measured by submitted code changes per week). We first compute Pearson’s correlation between MRR and properties mentioned above. However, the results are very small (absolute values are about 0.01). Then we turn to the relationship between MRR and  $DoR$ . We present the scatter plot of reviewer diversity and MRR in Figure 4. Overall, projects with low reviewer diversity can often obtain high MRR values. By contrast, projects with higher reviewer diversity obtain relatively low MRR. We compute the Pearson’s correlation between MRR and  $DoR$  for each evaluated approach. Correlation coefficients of REVFINDER, CHREV, COMMENT NETWORK and TIE are  $-0.37$ ,  $-0.48$ ,  $-0.42$  and  $-0.56$ , respectively. Overall, MRR and  $DoR$  are negatively correlated. We mine the reasons behind, shown below:

- 1) Nine projects engage only one or two reviewers. Therefore, all kinds of CRR approaches can achieve great scores.
- 2) Some projects (especially  $DoR$  values fall between 1.0 and 2.0) have multiple dominant reviewers. Yu *et al.* discovered in their qualitative study that on those projects with multiple dominant reviewers (*i.e.*, where multiple participants review code changes alternatively), expertise-based approaches achieve relatively low scores [16].
- 3) In some projects, many developers have multiple collaborators. For example, developer  $B$ ,  $C$  and  $D$  review 30% of code changes submitted by  $A$ , respectively, causing

it relatively difficult to determine which one is to be chosen.

**Finding 7.** Performance scores are overall negatively correlated to  $DoR$ .

1) *Performance Comparison:* Overall, COMMENT NETWORK outperforms other baseline approaches on proprietary projects. Specifically, COMMENT NETWORK achieve the higher scores than other three approaches on 137 out of 150 projects, with 13 projects in Others category excluded. As to open-source projects, Ouni *et al.* [13] evaluated their collaboration-based approach REVREC on three open-source projects, *i.e.*, Android, QT and OpenStack, and found REVREC outperforms other expertise-based approaches including REVFINDER, CHREV across a number of metrics. They also found that if REVREC only takes collaboration into account, the results are still acceptable. Therefore, it is not too strange that in the context of proprietary software, collaboration-based approach COMMENT NETWORK still performs much better. In our prior findings, there are over 50% of the reviewers have dominant collaborators, which means developers tend to select reviewers with whom they are familiar. It also indicates that proprietary software development may be similar to open-source development in some way. We also apply the Wilcoxon Signed Rank test [27] to our evaluation metrics (including accuracy, precision, recall and MRR) to confirm that the performance of COMMENT NETWORK is significantly better than other approaches, with all  $p$ -values lower than  $10^{-22}$ .

As is shown in Figure 1, 2 and 3, TIE achieves higher scores in terms of most metrics than CHREV and REVFINDER, except top-1 accuracy (equivalent to precision@1), which is not very important in practice [14]. We also utilize Wilcoxon Signed Rank test to confirm the performance metrics of TIE (except top-1 accuracy and precision@1) are significantly higher than file location-based approaches, with all  $p$ -values below  $10^{-20}$ .

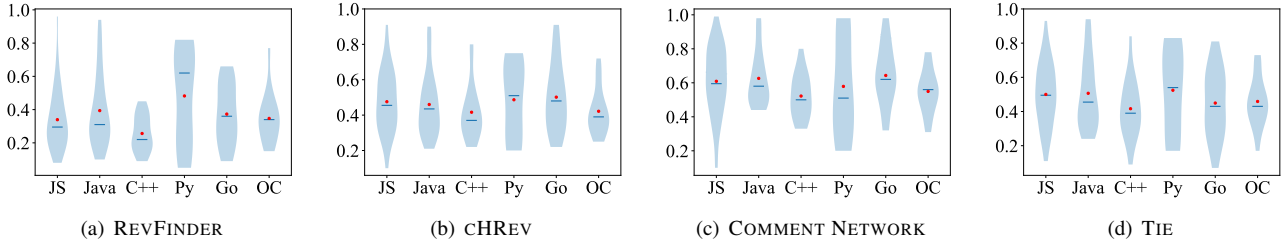


Fig. 2. Distribution of MRR of each group. Red points and blue lines indicate the averages and medians, respectively.

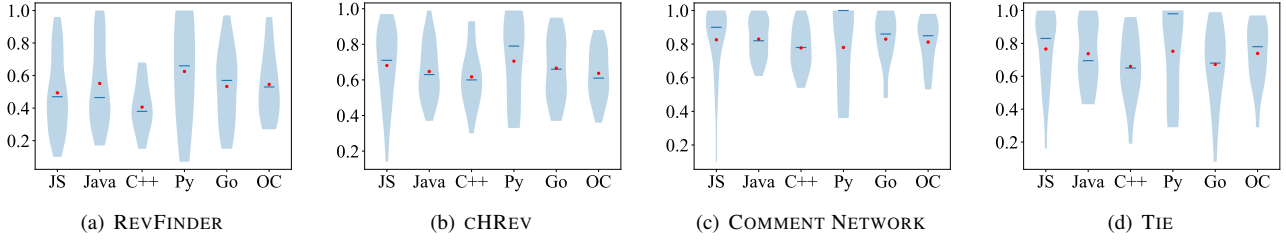


Fig. 3. Distribution of top-5 accuracy of each group. Red points and blue lines indicate the averages and medians, respectively.

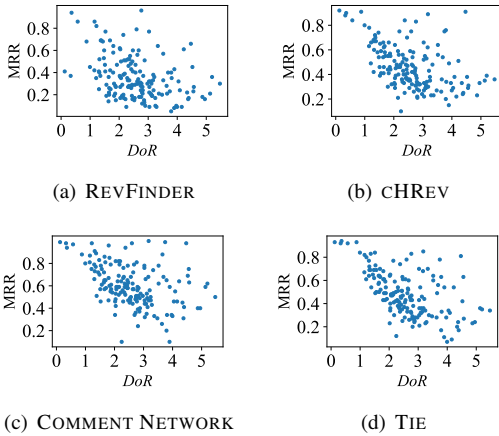


Fig. 4. Scatter plot of *DoR* and MRR.

**Finding 8.** Collaboration-based approach outperforms others by a large margin across most metrics, and text mining-based approach TIE completely outperforms file location-based approaches.

Another finding is that when REVFINDER is applied, the performance scores varies by programming languages more obviously. For example, we use Mann Whitney’s U test [28] to confirm that the MRR and top-5 accuracy values on C++ projects are significantly less than those projects written in other languages, with all  $p$ -values lower than 0.05. In the same way, performance metrics on Java projects are significantly lower than those on Python projects. This shows that the performance of REVFINDER is strongly subject to programming languages. By contrast, other approaches are not related to the

issue.

**Finding 9.** REVFINDER’s performance is more subject to programming languages, compared to other approaches.

## V. ANSWER RQ3: OUR APPROACH

### A. Insights

With help of what we gained from RQ1 and RQ2, we can conclude some directions and targets for our refined approach:

**Insight 1: We’d better consider collaboration of developers as a dominant factor.** On the one hand, we should follow the nature that developers tend to select reviewers they are familiar with. On the other hand, we notice that approaches such as REVFINDER and TIE may slow down the recommendation process while they cope with a small proportion of huge PRs. As a result, we may supplement developers’ expertise in a more lightweight way.

**Insight 2: We can take into account the multiple-project working experience of a proportion of reviewers.** Nearly 10% of the reviewers have multiple-project working experience. We may extract reviewers’ expertise from multiple projects. However, the content of reviewers of different projects are various. Therefore, it is beneficial to extract some general terms from identifiers, including file names and programming tokens in description of a pull request. We can further determine a reviewer’s expertise by extracted terms. Also, we need to consider collaboration in multiple projects, since close collaborators may also work together in another project.

### B. A New Approach to Better Recommend Reviewers

Our approach, namely CAMP, consists of *training* phase and *recommending* phase. In training phase, relative projects

can be trained together. CAMP builds a collaboration network from the collaboration history of a series of relative projects, and then divide the collaboration network into several communities using community detection algorithm. Then CAMP extracts terms from textual content and file paths of all PRs’ data in these relative projects, and generate a term list for each candidate reviewer. In recommending phase, CAMP first recommend the developers in the same community as the author by the weight of the edge between the developer and the author. If there are not enough reviewers, then CAMP extracts terms from the content of a given PR, and computes the relation score of the PR and each reviewer. Then reviewers with high relation scores are recommended.

1) *Training Phase*: Training phase comprises four steps, *i.e.*, constructing relation network, extracting technical terms, removing too common terms and generating term list for reviewers. Steps are described as follows.

**Step 1: Constructing relation network.** Like COMMENT NETWORK, CAMP also builds a relation network from a given project and its relative projects. Two members with collaboration share a undirected edge whose weight is the number of PRs they commonly participate in. If a developer has reviewed another reviewer’s code change, or the two developers worked in a common review, we say that they are *with collaboration*. Unlike COMMENT NETWORK, we use undirected graph instead, since the relationship between developers are two-way, *e.g.*, if *A* often review *B*’s code changes, then *B* will also review *A*’s from time to time. It then divides the network into some communities using *Louvain* algorithm [29] which is one of the widely used community detection approaches. Each community consists of a set of developers.

**Step 2: Extracting terms from pull requests.** We extract items from text and file paths of historical PRs in multiple relative projects. For textual content, we first remove stop words, URLs, email addresses and some unique identifiers, *e.g.*, Commit IDs, Pull Request IDs and related Issue IDs. We do not split words in English dictionary<sup>3</sup>. And for those words that are absent from dictionary, we utilize *Samurai* [30] algorithm to split them into soft words. However, we re-define *scoring function* in the splitting algorithm:

$$Score(t) = \alpha DocFreq(t) + (1 - \alpha) RevFreq(t) \quad (8)$$

where  $t$  is the given term,  $DocFreq(t)$  is the frequency of  $t$  in documentation website of corresponding project,  $RevFreq(t)$  is the frequency of  $t$  in review datasets, and  $\alpha$  is a coefficient, we set it 0.5 in experiments. We crawl HTML content from documentation websites, and only retain plain text. If there is no systematic documentation or the documentation is inaccessible, we ignore it.

For file paths, we split them by trailing slash into file and directory names. Then we also adopt *Samurai* algorithm to split file names into soft words, using the same scoring

function defined in Equation 8. After PRs’ content has been split into terms, we collect all terms to build a global term list.

**Step 3: Removing common terms.** If anyone of a community  $C$  has reviewed a code change containing the term  $t$ , we say that term  $t$  appears in community  $C$ . Intuitively, if a term appears in most communities, it is probably a general word. If a term appears only in few communities with high frequency, it is possibly a representative technology of a community.

For each term, CAMP computes the number of communities where it appears. We notice that terms such as `from`, `with`, and `commit` probably appear in almost all communities. In other words, these terms can hardly be used to distinguish proper reviewers since almost everyone uses them. Through our preliminary experiments, most terms in a PR appear in less than half of the communities, only a small proportion of terms appear in many communities. Therefore, we remove terms that appear in half of the communities, and finally the rest make up the term list.

**Step 4: Generating term list for each participant.** For each reviewers, we obtain the PRs in multiple projects he/she has participated in. Among all PRs he/she has participated in, We then count the appearing times of each term in term list obtain from Step 3. Finally this step produce a term list (with appearing time) for each participant.

2) *Recommending Phase*: In recommending phase, we first recommend reviewers in the same community as the author of a given PR. We rank the reviewers by the undirected weight between them and the author of the PR. If there are not enough reviewers, we then recommend reviewers by their expertise. We regard each incoming PR as a bag of terms, where terms are extracted in the training phase. Define  $S_R$  as the set of terms in a PR, and  $S_{Rev}$  as the term set related to a candidate reviewer. The confidence value between the incoming PR and a specific reviewer can be computed as follows:

$$Conf(R, Rev) = \sum_{t \in S_{Rev} \cap S_R} \frac{Count(t, R) \cdot Count(t, Rev)}{CommunityCount(t)} \quad (9)$$

where  $Count(t, R)$  is the times  $t$  appears in the review  $R$ , and  $Count(t, Rev)$  is the times  $t$  appears in the pull requests which have been reviewed by the reviewer  $Rev$ .  $CommunityCount(t)$  denotes the number of community where the term  $t$  appears, as defined in Step 3.

### C. Evaluation

We evaluate CAMP on our datasets described in Section III-A using the same performance metrics and baseline approaches listed in Section IV-B. Due to space limitation, we only present the average top-k accuracy, precision and recall in Figure 5 and distribution of MRR and top-5 accuracy in Figure 6, respectively. Overall, CAMP outperforms other baseline approaches across almost all metrics. From Figure 6(a), CAMP achieves significantly higher MRR on JavaScript, Java and Go projects than COMMENT NETWORK. As is shown

<sup>3</sup>We use the dictionary in <https://github.com/dwyl/english-words>



## VI. DISCUSSION

### A. Directions for Code Reviewer Recommendation

in Figure 6(b), there are less projects that only receive very low top-5 accuracy when recommended by CAMP, that is, CAMP produces acceptable recommendation results for more projects. Specifically, there are about 21% of the projects that only receive top-5 accuracy lower than 0.7 when COMMENT NETWORK is adopted, while only 10% when CAMP is applied. Similar to Section IV, we utilize Wilcoxon Signed Rank test [27], and also compute Cliff’s Delta [31] to confirm that CAMP performs significantly better than other approaches.

CAMP outperforms state-of-the-art approaches, and can produce acceptable results for more projects.

We further compute the proportion of code changes that receive more accurate recommendation. The results show that nearly 20% pull requests get more accurate recommendation on average, up to 29%. We pay attention to those pull requests with a large number of files as well, and discover that over 50% of such PRs get more accurate recommendation with our approach CAMP. These findings also prove the effectiveness of our approach as well.

We also quantitatively analyze two kinds of projects: (i) projects that the traditional CRR approaches cannot satisfactorily cope with, but CAMP can produce acceptable results, (ii) projects on which CAMP performs much better. We find that for many code changes, CAMP can recommend more participants close to the contributors, and these participants did not often collaborate with the contributors in the past but have collaborated in other projects. This can confirm the effectiveness of considering multiple-project working experience.

Besides, we use *reviewer coverage* to measure the participation rate of reviewers. CAMP can improve reviewer coverage of COMMENT NETWORK, REV FINDER, CHREV and TIE by about 10% (from 73% to 81%), 55% (from 52% to 81%), 8% (from 75% to 81%) and 8% (from 75% to 81%), respectively. CAMP tends to recommend reviewers in the same community and have relative expertise (e.g., having worked on files with a specific token many times). This proves the effectiveness of extracting terms from the content of pull requests. We define reviewer coverage (*RC*) as follows:

$$RC(P, Rec) = \frac{\text{card}(\bigcup_{pr \in P} Rec(pr, 10))}{\#Rev(P)} \quad (10)$$

where  $P$  denotes a certain project,  $Rec$  stands for a recommendation approach and thus  $Rec(pr, k)$  denotes the first  $k$  recommended reviewers for  $pr$ , and  $\#Rev(P)$  indicates the total number of reviewers in  $P$ .

CAMP can recommend more diverse reviewers, as a result, increasing the participation rate.

**Better goals should be set for CRR approaches.** Through our exploratory study and evaluation of our proposed approach, we confirm that our approach can achieve acceptable performance scores on almost all projects. However, we should also lay stress not only on traditional performance metrics, but also on whether reviewer recommenders can distribute knowledge, promote team awareness, etc. Through our evaluation, our approach CAMP outperforms others in *reviewer coverage*, which means CAMP can increase the participation rate of internal developers. We believe it is what our approach can benefit internal development.

**A wide range of external knowledge should be mined to better recommend reviewers.** Terms in a text and file paths may relate to a wide range of external knowledge, such as documentation of the project and programming conventions. In this work, many proprietary projects lack relative documentation or their documentation is inaccessible. We notice that CARROT [20], a context-aware recommendation approach leverages some metadata, including directory, repositories and file extensions. Some recommendation systems in Software Engineering tasks, such as [32], [33] make use of knowledge from technical websites, such as StackOverflow. Additionally, we may utilize knowledge graph (KG) to better organize a wide range of information when recommending proper reviewers. Recently knowledge graph has been applied in many software engineering tasks, e.g., API recommendation, bug localization.

### B. Limitation and Threats to Validity

One limitation of CAMP is that it cannot fully outperform COMMENT NETWORK on a small part of projects with a large number of pull requests. The scores in terms of top- $k$  accuracy and precision when  $k$  are relatively small. It indicates that for those projects, the expertise of reviewers are difficult to determine, since pull requests contain too much noise, thus considering expertise factors may lower the performance. On the contrary, although recommending reviewers only by their collaboration cannot achieve very high scores in terms of most metrics, it can yet obtain very high performance on a proportion of projects.

Another limitation of our study is that we do not investigate the universality of our findings and proposed approaches. Although we do empirical study and evaluate our new approach on up to 163 proprietary projects to mitigate threats to external validity, our findings and approach may not be suitable for other proprietary projects in Tencent and other commercial companies. Also, we do not collect many relevant open-source projects, thus we cannot check whether CAMP is suitable for OSS development. It is left for future studies to investigate how CRR approaches perform proprietary projects in different commercial software companies.

There may also be some errors and biases in our datasets and experiments. First, in Tencent, contributors may make

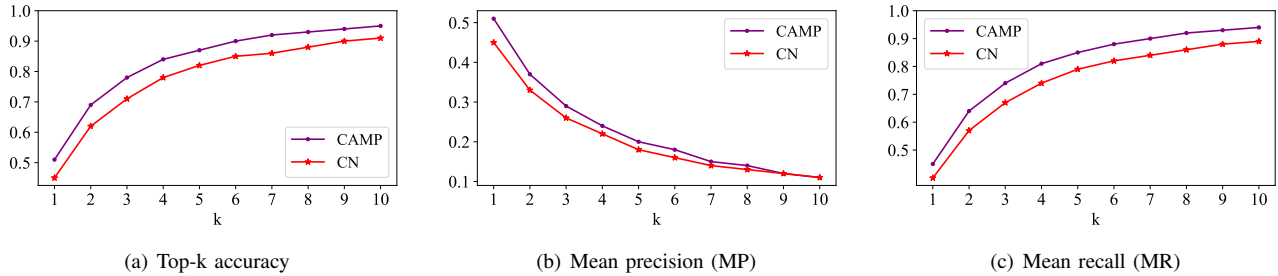


Fig. 5. Performance of CAMP, compared to COMMENT NETWORK.

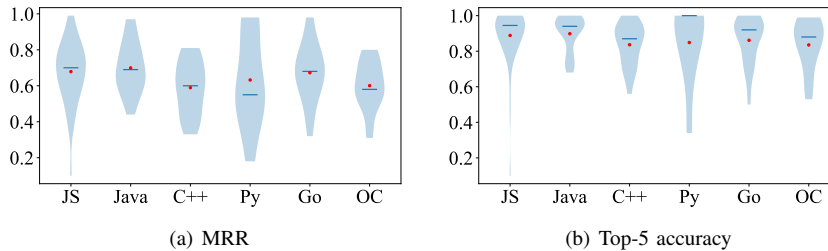


Fig. 6. Distribution of MRR and top-5 accuracy of CAMP. Red points and blue lines indicate the averages and medians, respectively.

wrong decisions or only receive the default choices when selecting reviewers. Second, we do not collect code changes contributed or reviewed by robots, which may ignore technical terms related to some developers. Also, the identifier splitting method might wrongly cut out-of-vocabulary words, causing errors in computing similarity between pull requests.

## VII. RELATED WORK

Recommendation systems are also widely used in many other software engineering tasks as well as code reviewer selection, *e.g.*, library recommendation, API recommendation, and code snippet recommendation. We mainly concern the first two in this section.

For library recommendation, many approaches leverage collaborative filtering, heuristic search that is used in some CRR approaches. For example, Thung *et al.* proposed *LibRec* [34] to automatically recommend relative libraries for developers using both associate rule mining and collaborative filtering. Ouni *et al.* formulated library recommendation as a multi-objective search problem [35]. Chen *et al.* proposed an approach to mine analogue libraries base on the knowledge base from StackOverflow [33]. Recently Nguyen *et al.* proposed CROSSREC [36] to recommend third-party libraries also using collaborative filtering.

For API recommendation, McCarey *et al.* recommend methods to a developer in a group of developers by investigating the historical methods that the developers have used [37]. Chan *et al.* propose a new approach that recommends API methods by text phrases [38]. Thung *et al.* proposed an API recommendation framework [39] based on both history-based and description-based recommenders. They also proposed an approach WEBAPIREC [32] dedicated to automatically recommending that takes as input a project profile and outputs

a ranked list of web APIs that can be used to implement the project. Moreno *et al.* proposed MUSE to recommend code examples by extracting concrete method usages [40].

## VIII. CONCLUSION AND FUTURE WORK

Code Reviewer Recommendation (CRR) is an important task for modern software development. Since a large number of submitted code changes, it is necessary to automate CRR to improve development efficiency. In this paper, we first conduct an empirical study on a large scale of proprietary projects from Tencent. We both quantitatively and qualitatively illustrate the characteristics of proprietary projects and the effectiveness of existing CRR approaches. Based on our findings and implications, we propose a new method CAMP that recommends reviewers according to their collaboration and expertise on multiple projects. We evaluate our proposed approach, and the results show that CAMP can obtain higher scores than other baseline approaches across most performance metrics. Additionally, CAMP can mitigate diversity of performance scores on proprietary projects, *i.e.*, offer acceptable performance for more projects. We hope our study can provide insights for both researchers and practitioners.

In the future, we plan to integrate our new approach in development platforms in Tencent and investigate how CAMP works in practice. We also plan to conduct a further study to deeply understand the developers' perception and advice of CRR to meet practitioners' expectations.

## ACKNOWLEDGMENT

This research is supported by the National Science Foundation of China (No. U20A20173 and No. 6190234).

## REFERENCES

- [1] M. Fagan, "Design and code inspections to reduce errors in program development," in *Software pioneers*. Springer, 2002, pp. 575–607.
- [2] A. Aurum, H. Petersson, and C. Wohlin, "State-of-the-art: software inspections after 25 years," *Software Testing, Verification and Reliability*, vol. 12, no. 3, pp. 133–154, 2002.
- [3] A. F. Ackerman, P. J. Fowler, and R. G. Ebenau, "Software inspections and the industrial production of software," in *Proc. of a Symposium on Software Validation: Inspection-Testing-Verification-Alternatives*. USA: Elsevier North-Holland, Inc., 1984, p. 13–40.
- [4] W. H. A. Al-Zubaidi, P. Thongtanunam, H. K. Dam, C. Tantithamthavorn, and A. Ghose, "Workload-aware reviewer recommendation using a multi-objective search-based approach," in *Proceedings of the 16th ACM International Conference on Predictive Models and Data Analytics in Software Engineering*, 2020, pp. 21–30.
- [5] F. Shull and C. Seaman, "Inspecting the history of inspections: An example of evidence-based technology diffusion," *IEEE Software*, vol. 25, no. 1, pp. 88–90, 2008.
- [6] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 202–212.
- [7] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 712–721.
- [8] X. Xia, D. Lo, X. Wang, and X. Yang, "Who should review this change?: Putting text and file location analyses together for more accurate recommendations," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 261–270.
- [9] V. Mashayekhi, J. M. Drake, W.-T. Tsai, and J. Riedl, "Distributed, collaborative software inspection," *IEEE software*, vol. 10, no. 5, pp. 66–75, 1993.
- [10] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 931–940.
- [11] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto, "Who should review my code? a file location-based code-reviewer recommendation approach for modern code review," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 141–150.
- [12] M. B. Zanjani, H. Kagdi, and C. Bird, "Automatically recommending peer reviewers in modern code review," *IEEE Transactions on Software Engineering*, vol. 42, no. 6, pp. 530–543, 2015.
- [13] A. Ouni, R. G. Kula, and K. Inoue, "Search-based peer reviewers recommendation in modern code review," in *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*. IEEE Computer Society, 2016, pp. 367–377. [Online]. Available: <https://doi.org/10.1109/ICSME.2016.65>
- [14] M. M. Rahman, C. K. Roy, and J. A. Collins, "Correct: code reviewer recommendation in github based on cross-project and technology experience," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 222–231. [Online]. Available: <https://doi.org/10.1145/2889160.2889244>
- [15] J. Kim and E. Lee, "Understanding review expertise of developers: A reviewer recommendation approach based on latent dirichlet allocation," *Symmetry*, vol. 10, no. 4, p. 114, 2018.
- [16] Y. Yu, H. Wang, G. Yin, and T. Wang, "Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment?" *Information and Software Technology*, vol. 74, pp. 204–218, 2016.
- [17] H. Ying, L. Chen, T. Liang, and J. Wu, "Earec: leveraging expertise and authority for pull-request reviewer recommendation in github," in *2016 IEEE/ACM 3rd International Workshop on CrowdSourcing in Software Engineering (CSI-SE)*. IEEE, 2016, pp. 29–35.
- [18] E. Mirsaedi and P. C. Rigby, "Mitigating turnover with code review recommendation: balancing expertise, workload, and knowledge distribution," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1183–1195.
- [19] A. Chueshev, J. Lawall, R. Bendraou, and T. Ziadi, "Expanding the number of reviewers in open-source projects by recommending appropriate developers," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 499–510.
- [20] A. Strand, M. Gunnarson, R. Britto, and M. Usman, "Using a context-aware approach to recommend code reviewers: findings from an industrial case study," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, 2020, pp. 1–10.
- [21] S. Asthana, R. Kumar, R. Bhagwan, C. Bird, C. Bansal, C. Maddila, S. Mehta, and B. Ashok, "Whodo: automating reviewer suggestions at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 937–945.
- [22] S. Rebai, A. Amich, S. Molaei, M. Kessentini, and R. Kazman, "Multi-objective code reviewer recommendations: balancing expertise, availability and collaborations," *Automated Software Engineering*, vol. 27, no. 3, pp. 301–328, 2020.
- [23] C. E. Shannon, "A mathematical theory of communication," *The Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.
- [24] R. Agrawal, R. Srikant et al., "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215. Citeseer, 1994, pp. 487–499.
- [25] R. Baeza-Yates, B. Ribeiro-Neto et al., *Modern information retrieval*. ACM press New York, 1999, vol. 463.
- [26] Y. Hu, J. Wang, J. Hou, S. Li, and Q. Wang, "Is there a "golden" rule for code reviewer recommendation?:—an experimental evaluation," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2020, pp. 497–508.
- [27] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics*. Springer, 1992, pp. 196–202.
- [28] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.
- [29] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, 2008.
- [30] E. Enslin, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," in *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 71–80.
- [31] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological bulletin*, vol. 114, no. 3, p. 494, 1993.
- [32] F. Thung, R. J. Oentaryo, D. Lo, and Y. Tian, "WebapiREC: Recommending web apis to software projects via personalized ranking," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 1, no. 3, pp. 145–156, 2017.
- [33] C. Chen, S. Gao, and Z. Xing, "Mining analogical libraries in q&a discussions—incorporating relational and categorical knowledge into word embedding," in *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 338–348.
- [34] F. Thung, D. Lo, and J. Lawall, "Automated library recommendation," in *2013 20th Working conference on reverse engineering (WCRE)*. IEEE, 2013, pp. 182–191.
- [35] A. Ouni, R. G. Kula, M. Kessentini, T. Ishio, D. M. German, and K. Inoue, "Search-based software library recommendation using multi-objective optimization," *Information and Software Technology*, vol. 83, pp. 55–75, 2017.
- [36] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, and M. Di Penta, "Crossrec: Supporting software developers by recommending third-party libraries," *Journal of Systems and Software*, vol. 161, p. 110460, 2020.
- [37] F. McCarey, M. Ó. Cinnéide, and N. Kushmerick, "Rascal: A recommender agent for agile reuse," *Artificial Intelligence Review*, vol. 24, no. 3-4, pp. 253–276, 2005.
- [38] W.-K. Chan, H. Cheng, and D. Lo, "Searching connected api subgraph via text phrases," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.
- [39] F. Thung, S. Wang, D. Lo, and J. Lawall, "Automatic recommendation of api methods from feature requests," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 290–300.

[40] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How can i use this method?" in *2015 IEEE/ACM 37th IEEE International*

*Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 880–890.